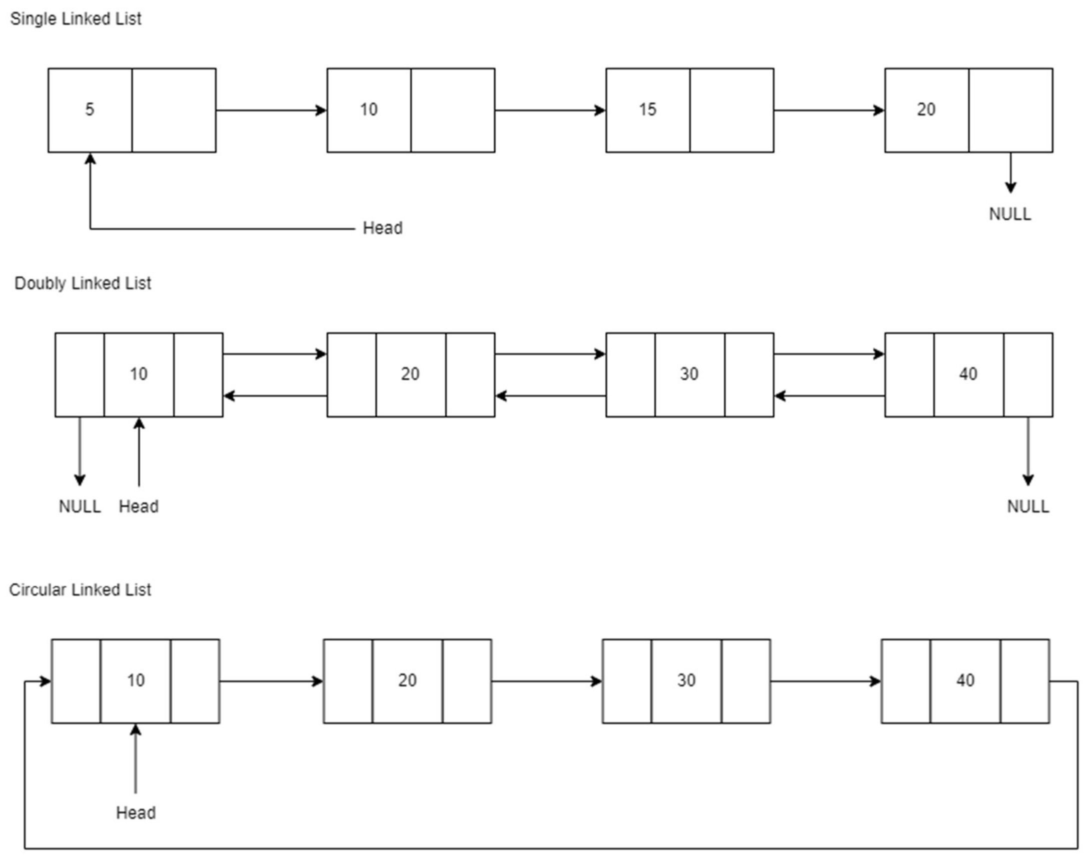
LINKED LIST

// number 1



// number 2

There are several differences between arrays and linked lists. Regarding the type of memory used, arrays utilize static memory, while linked lists utilize dynamic memory. This means that the memory used by arrays are fixed, contrasting to the resizable memory of linked lists. Because of this, deletion and insertion are time-consuming in arrays, unlike the quicker pace of linked lists.

Additionally, arrays store elements in consecutive memory addresses while linked lists store their members in a scattered manner. Arrays are random access structures (using indexing), while linked lists utilize sequential access (traverse in order). Therefore, searching in arrays are more efficient than in linked lists. Aside from that, linked lists require more storage than arrays because unlike arrays, each node also contains pointer(s).

// number 3

Floyd’s Algorithm is an algorithm used to find the shortest path between all pairs of vertices in a weighted graph.

STACK AND QUEUE

// number 1

There are several differing things regarding stack and queue. Firstly, both data structures use differing approaches. Stack uses the Last in First Out approach (LIFO), while queue uses the First In First Out approach (FIFO). While both use similar operations, they use different terms. Insertion and deletion in stack are called push and pop respectively, however they are known as enqueue and dequeue in queue data structures. Push and pop happen at one end of the stack. In comparison, enqueue and dequeue take place at opposing ends of the queue.

// number 2

In prefix notation, every operator is written before two operands.

Stack implementation:

```
    Scan string from right to left for each character
        If an operand is found
            Push to stack
        If an operator is found
            Pop the top two elements (store it in variable A and B in that order)
            Push the result (A operator B)
```

In infix notation, every operator is written between two operands.

Stack implementation:

```
    Evaluate the infix expression
    Search the highest precedence operator
    For each search, traverse the string for every operator
```

In postfix notation, every operator is written after two operators.

Stack implementation:

```
    Scan string from left to right for each character
        If an operand is found
            Push to stack
        If an operator is found
            Pop the top two elements (store it in variable A and B in that order)
            Push the result (B operator A)
```

HASHING AND HASH TABLES

// number 1

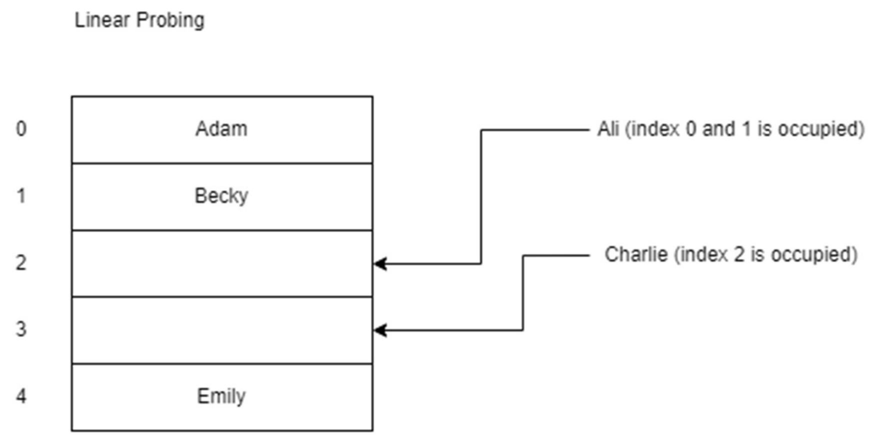
Hash tables are data structures usually used to store strings in an associative manner. In the hash table, hashing, which is the process where a string is converted into a hashed string, takes place. The index of the hash table indicates the hashed string, while the related value is the original string.

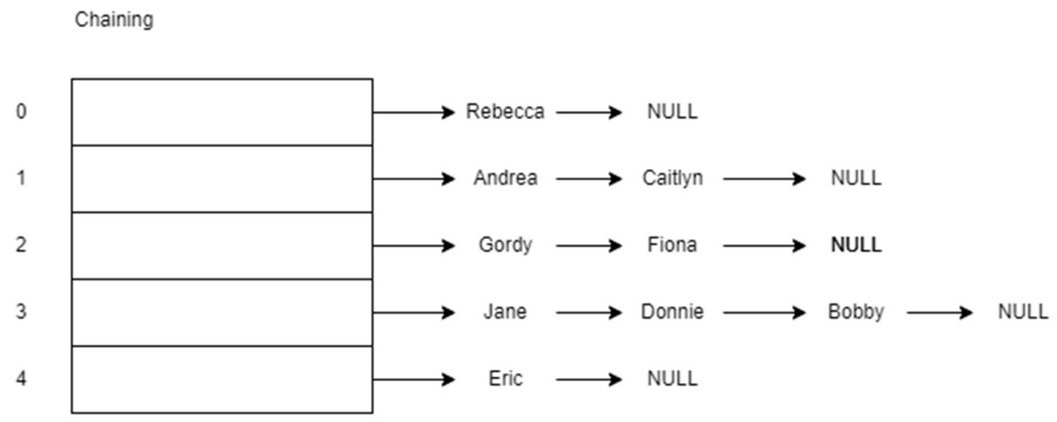
Hash functions are functions used to hash, or convert a string in various ways.

Collision handling is a way to handle situations where a new key is mapped to an already occupied slot in the hash table, such as linear probing and chaining.

// number 2

There are two methods for collision handling, which are linear probing and chaining.


Linear Probing
0
Adam
1
Becky
2
3
4
Emily
Ali (index 0 and 1 is occupied)
Charlie (index 2 is occupied)

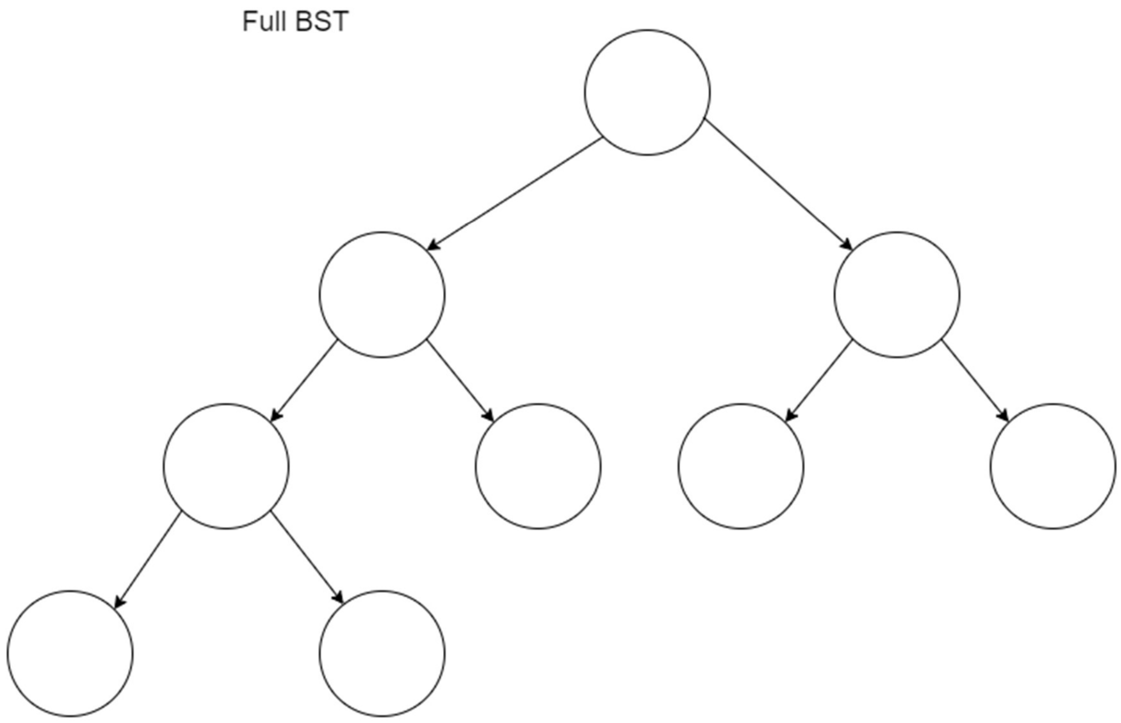


BINARY SEARCH TREE

// number 1

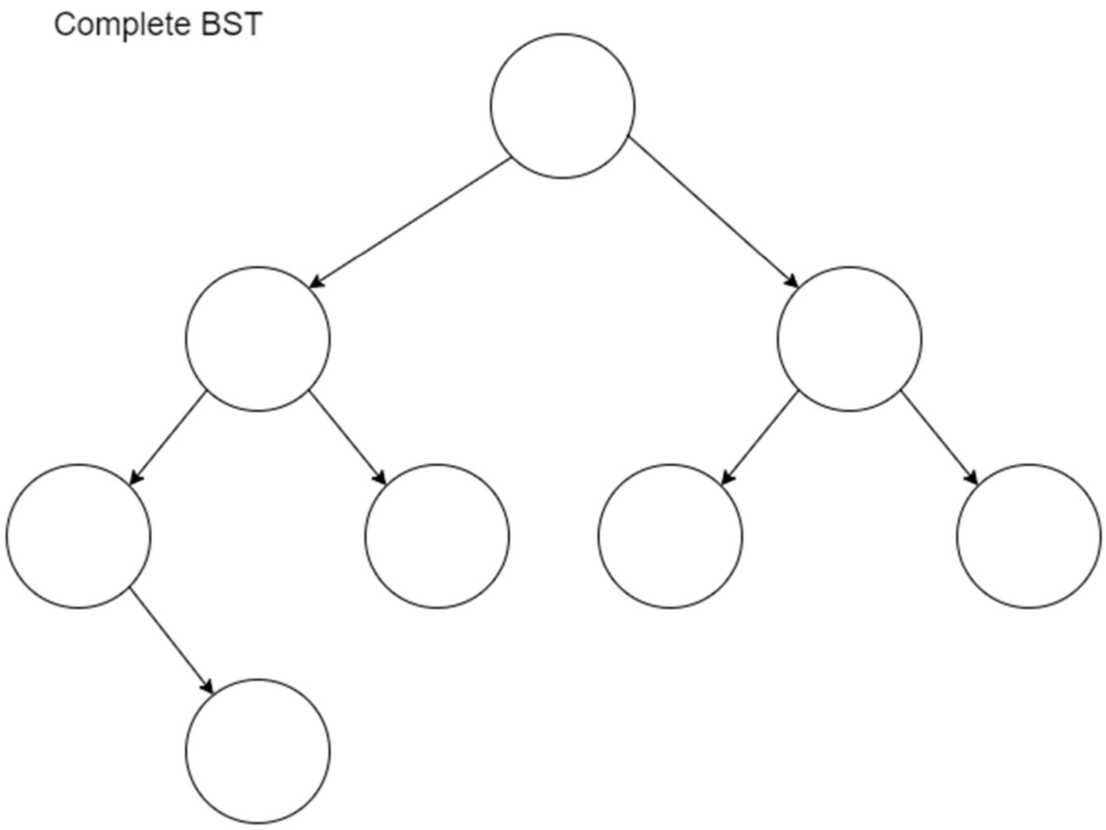
Full Binary Search Tree

BST where each node has either no children or two children



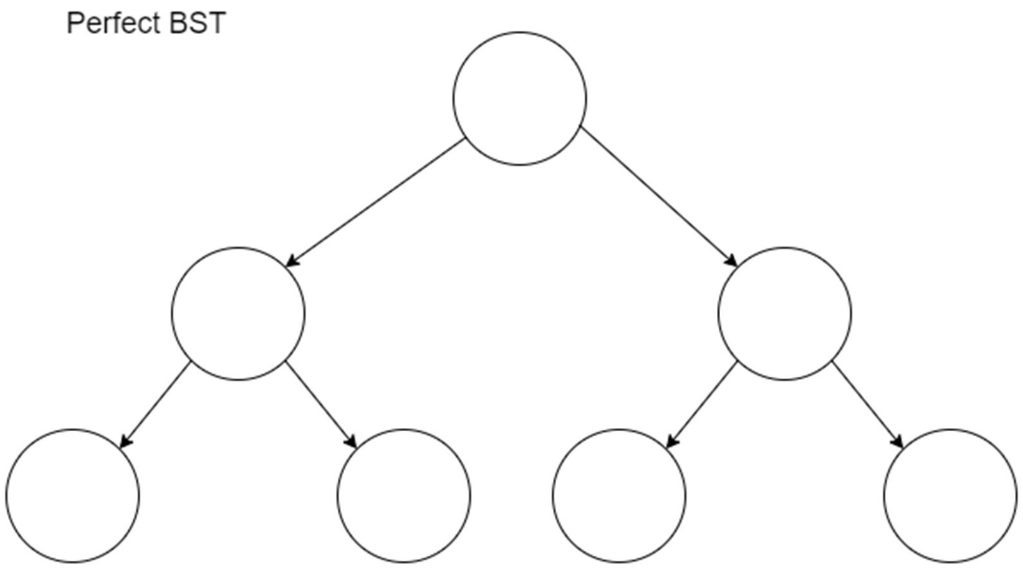
Complete Binary Search Tree

BST where all tree levels are filled entirely except the lowest level



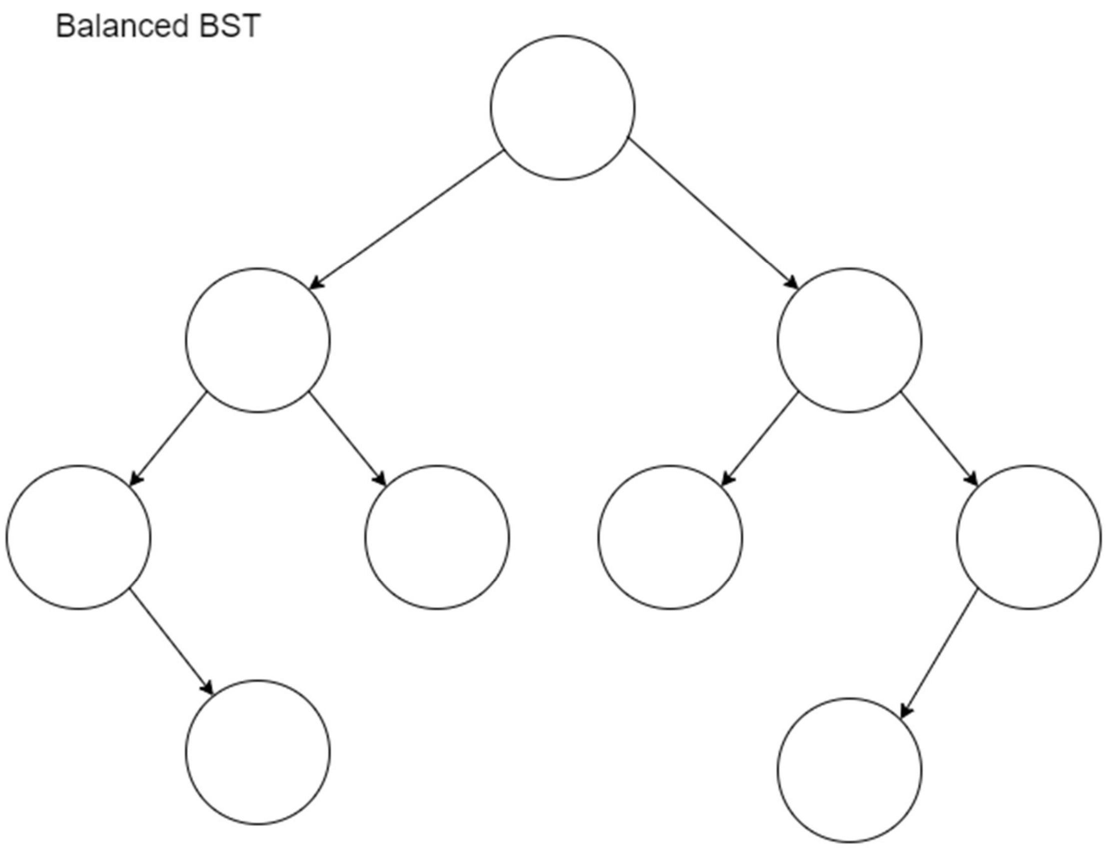
Perfect Binary Search Tree

BST where each node has strictly two children and every leaf node is at the same level



Balanced Binary Search Tree

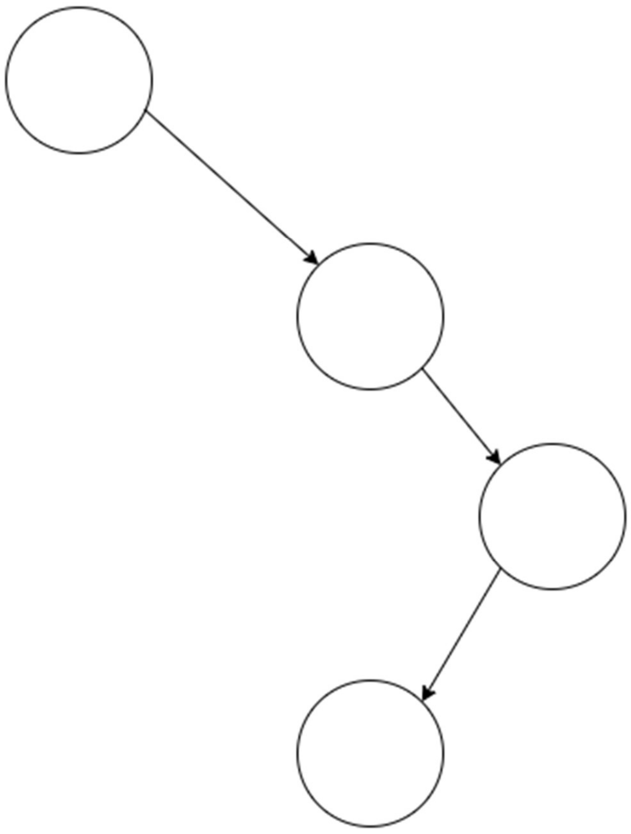
Height of the left and right subtree vary at most one



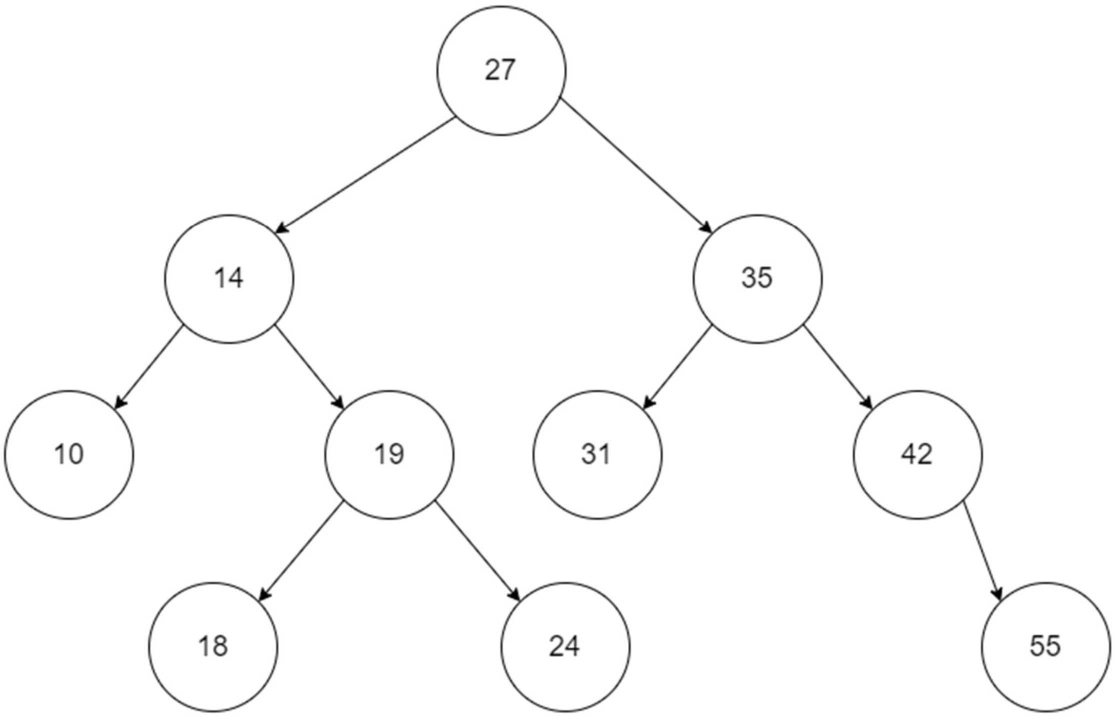
Degenerate Binary Search Tree

BST where every internal node has one child, similar to a linked list

Degenerate BST



// number 2



Insert 24

24 < 27, so go to left subtree

24 > 14, so go to right subtree

24 > 19, so go to right subtree

Node == NULL, so create new node

Insert 18

18 < 27, so go to left subtree

18 > 14, so go to right subtree

18 < 19, so go to left subtree

Node == NULL, so create new node

Insert 55

55 > 27, so go to right subtree

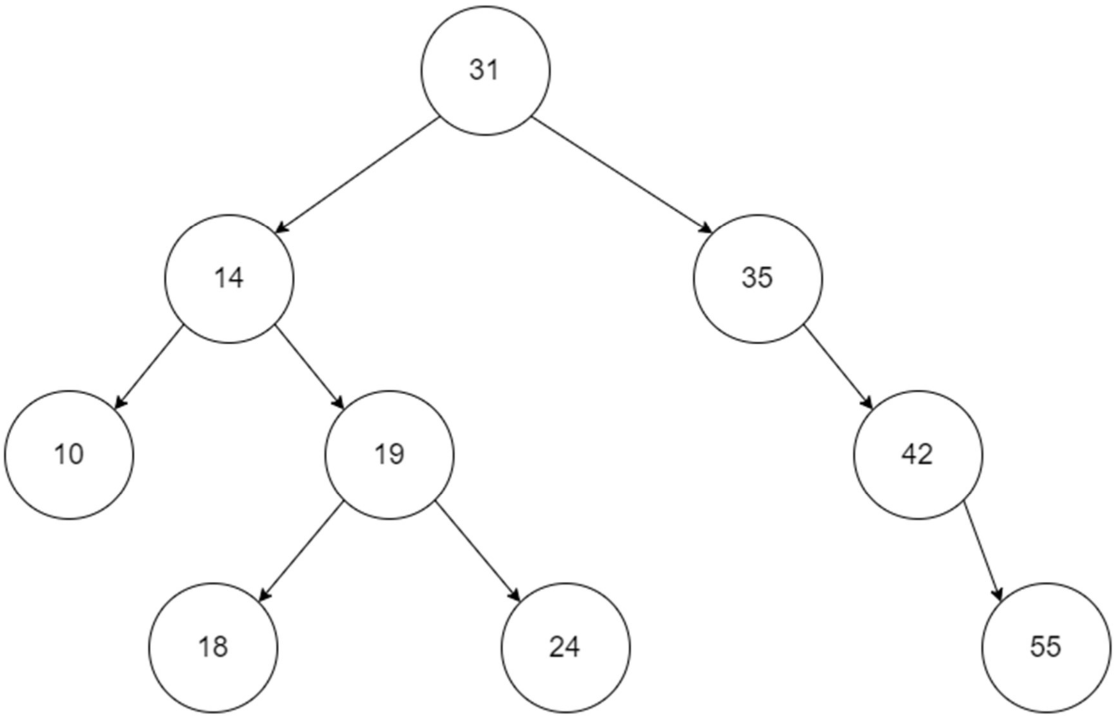
55 > 35, so go to right subtree

55 > 42, so go to right subtree

Node == NULL, so create new node

// number 3

Delete 27

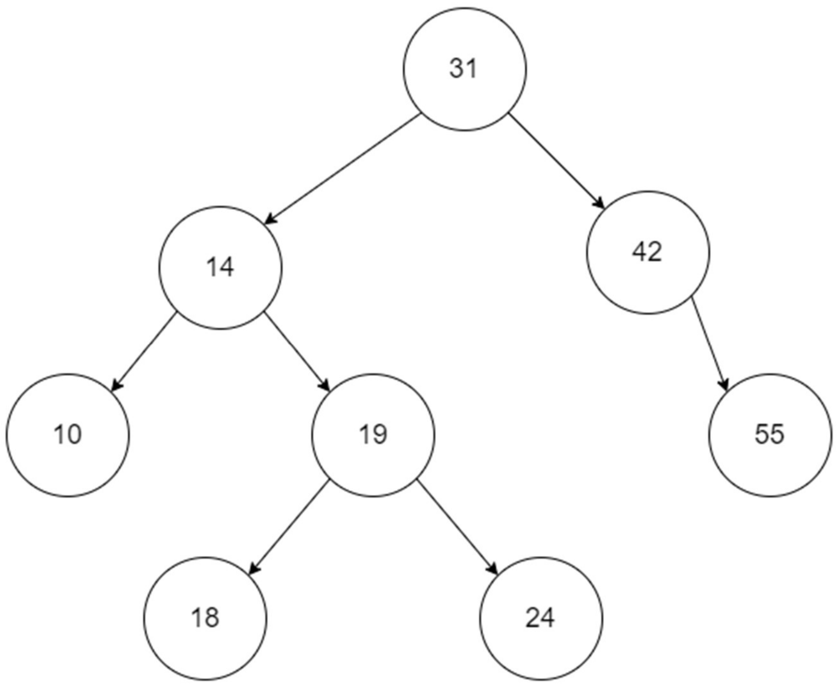


Find in-order successor of 27 which is 31

Remove 27 by replacing it with 31

Remove previous 31 node

Delete 35

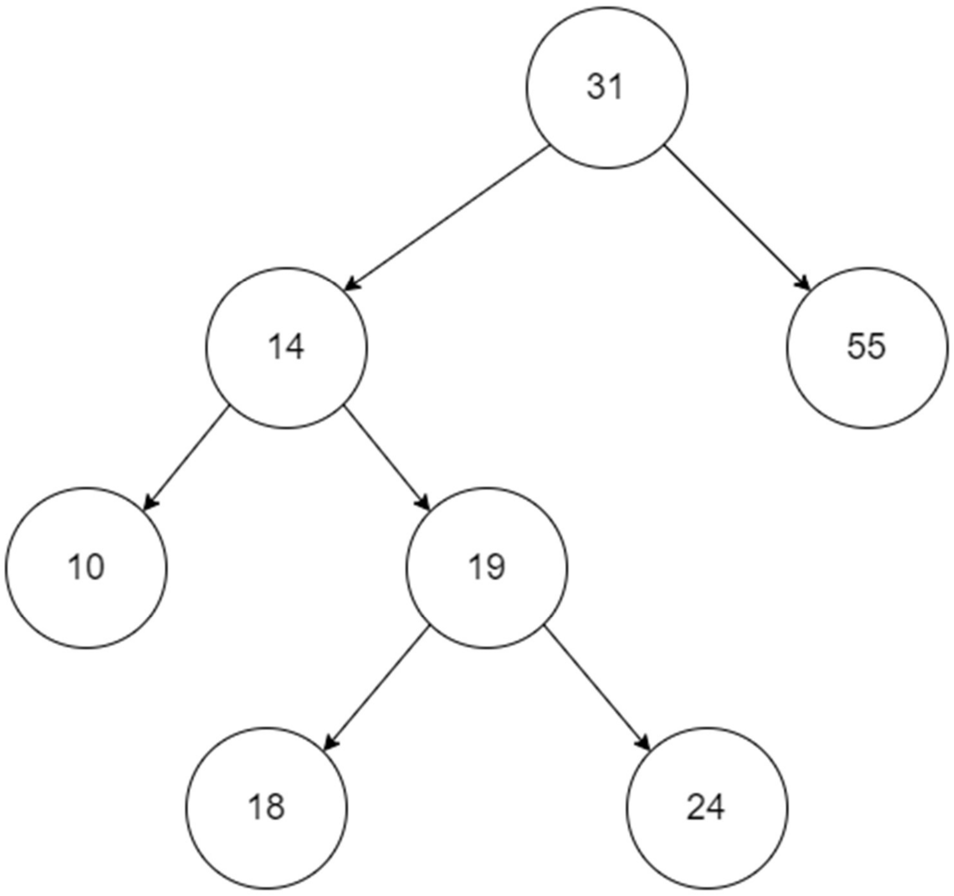


Find in-order successor of 35, there is none, so candidate node is 42

Remove 35 by replacing it with 42

Remove previous 42 node

Delete 42



Find in-order successor of 42, there is none, so candidate node is 55

Remove 42 by replacing it with 55

Remove previous 55 node